Raj Patel, Derek Sagun, Luiz River

PA3 Report

In order to run our program:

First create a mount point inside of /tmp. For an example, you can run the command “mkdir /tmp/mounty” to create a folder called mounty inside of tmp which we will use as the mount point.

Secondly, go into the ServerSNFS and run the make command. The server executable follows the same structure as specified in the assignment document, so you can run the server as such: “./serverSNFS -port 12457 -mount /tmp/mounty” After the server runs, it should print out the IP address of the host.

Then, go into the clientSNFS and run the make command. Afterwards, execute the following command “./ssfs -f /tmp/mounty” , including the name of the mount path after -f.

Then, the program will prompt input for the port number, which should match the argument you gave to the server, and IP address, which should match the IP address printed out by the server.

After this, the connection will be made between the client and the server, this will be noticeable as a print statements will appear stating the connection has been made.

In order to test our program:

We have included a folder called test cases, which has a bunch of programs that test a lot of the function calls. We will list each function and describe how to test each one. We have created a program to test the following functions: **create, open, read, write, truncate**. Within each of the functions, we also open and close the files accordingly thus, **flush and release** are also tested. In order to test the following: **readDir, openDir, relaseDir** and **getAttr** again, we can run “dir” on the directory to print out its contents. Lastly, we can test the **mkdir** function by explicitly running “mkdir /tmp/mounty/newDir” or something similar. Next we will describe the command line structure for parameters for the programs in testcases. For each program, you can enter the directory and run make.

Here are the test cases we would like to run:

testCreate: takes in path of the file to create appended to the mount path

Run the following commands (after running make in terminal of testCreate directory):

./testCreate /tmp/mounty/newFile

./testCreate /tmp/mounty/newFile2.txt

./testCreate /tmp/mounty/newFile3.pdf

This creates 3 new files in the server’s mount path, all with the S\_IRWXU permission

The testCreate program will print out the file descriptor of the file that was returned by the command. Closes the file afterwards

testOpen: takes in path of the file to open appended to the mount path

Run the following commands (after running make in terminal of testOpen directory):

./testOpen /tmp/mounty/newFile3.pdf

./testOpen /tmp/mounty/newFile2.txt

Opens these two files and returns file descriptors, closes the files after use

testWrite: takes in path of file to write to appended to mount path, and a string to write into the file (MAX 50 bytes)

Run the following commands (after running make in terminal of testWrite directory):

./testWrite /tmp/mounty/newFile “test1 test1 test1”

./testWrite /tmp/mounty/newFile2.txt “this is the second test of write”

This writes the strings in quotes to the following files. To test this, you can run

cat /tmp/mounty/newFile or cat /tmp/mounty/newFile2.txt

or you can also just open the files manually

testRead: takes in path of the file to read from appended to mount path (tries to read 50 bytes)

Run the following commands (after running read in terminal of testRead directory):

./testRead /tmp/mounty/newFile

./testRead /tmp/mounty/newFile2.txt

This will print out the first 50 bytes of the file at the path.

testTruncate: takes in path of the file to truncate and the size to truncate the file to

Run the following command

./testTruncate /tmp/mounty/newFile2.txt 5

This will truncate the file to 5 bytes. This can be examined by clicking the file and examining its size.

For all the functions above except for truncate, the file is CLOSED at the end of every test program; thus flush & release are called at the end of testCreate, testOpen, testRead, and testWrite.

In order to test getAttr, readDir, openDir, & releaseDir, we can run the terminal command:

dir /tmp/mounty

This will print out the contents of the directory and list all of the following files.

Cat and Dir are terminal programs that work for our implementation of FUSE

Responsibilities:

Raj : Open, Truncate, Create , Wrote testCreate, testOpen

Derek: Read, Write, Flush, Release, Wrote testRead, testWrite, testTruncate

Luiz: MkDir, Readdir, ReleaseDir, GetAttr, OpenDir

We created a FUSE File system that implements all of the functions asked for in the assignment document (open, create, read, write, flush, release, truncate, opendir, readdir, mkdir, relasedir, getattr). Our RPC involves a system of codes that correspond to each function, along with the path, since essentially every single FUSE function needs the path of the file or directory. For instance, the code for Open is 22. So if an open function was called on the mount path, FUSE would send “22/mountpath/file” to the server. The server would examine the first two characters and call the handle open function, which just awaits the specific parameters for the open function (the flags) and sends back confirmation. After receiving the parameters from the client, the server executes the system call and sends back the result, possibly with an errno if the system call fails. Most of the file call implementations work in this manner. Some difficulties we came across included our initial idea for RPC which was to send structs over a socket. However, this did not work, and we decided to go the simple manner of just sending a code and awaiting parameters for each individual function. Our server is also multithreaded and can handle multiple clients, as long as they make requests to operate on different files. Another challenge we faced was determining which calls were dependent on which. For an example, I had tried to implement the create function first, but I was unsuccessful, and it took me some time to realize that create actually called get\_attr as well as open before actually routing to the create call. These dependencies proved to be a challenge since they could only be tested after most of the calls were complete.